AIMLZG537
Information Retrieval

Dr. Maheswari Karthikeyan

Lecture4

BITS Pilani
Pilani Campus

Lecture Outline

innovate achieve lead

Chapter 3: Dictionaries and Tolerant Retrieval

- Dictionary Data Structures
- Tolerant Retrieval

Slides are adapted from Introduction to Information Retrieval;Manning,Raghavan

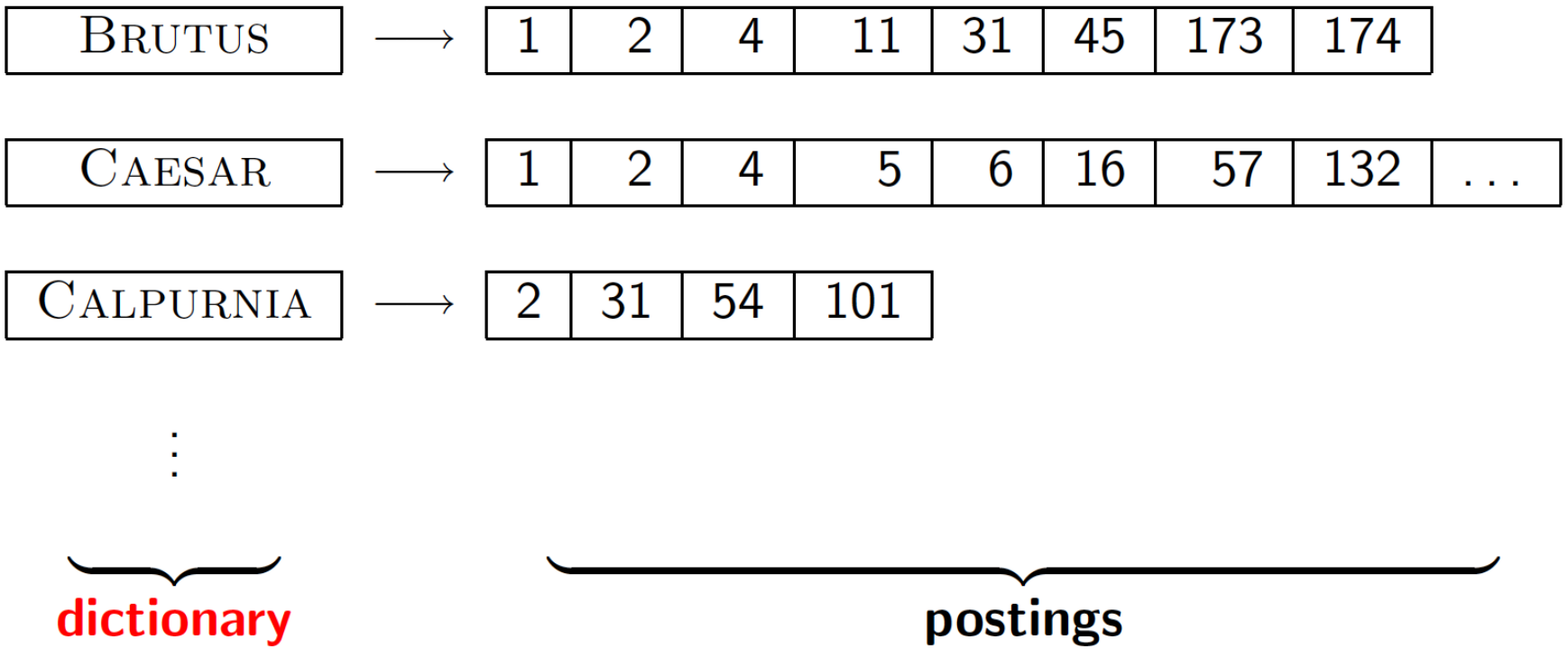
http://nlp.stanford.edu/IR-book

Dictionaries and Tolerant Retrieval

- **Dictionary data structures**
  - data structures that help the search for terms in the vocabulary in an inverted index.
- **Tolerant retrieval**
  - techniques that are robust to typographical errors in the query, as well as alternative spellings.
  - Wild-card queries
  - Spelling correction
  - Phonetic correction

Search structures for dictionaries

- The dictionary stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?



innovate achieve lead

A simple dictionary

- An array of structures:

term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

char[20] int Postings

20 bytes 4/8 bytes 4/8 bytes

Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees

The choice of solution (hashing, or search trees) depends on:

- No. of keys(Terms)
- Keys remain static or dynamic
- The relative frequencies with which various keys will be accessed.

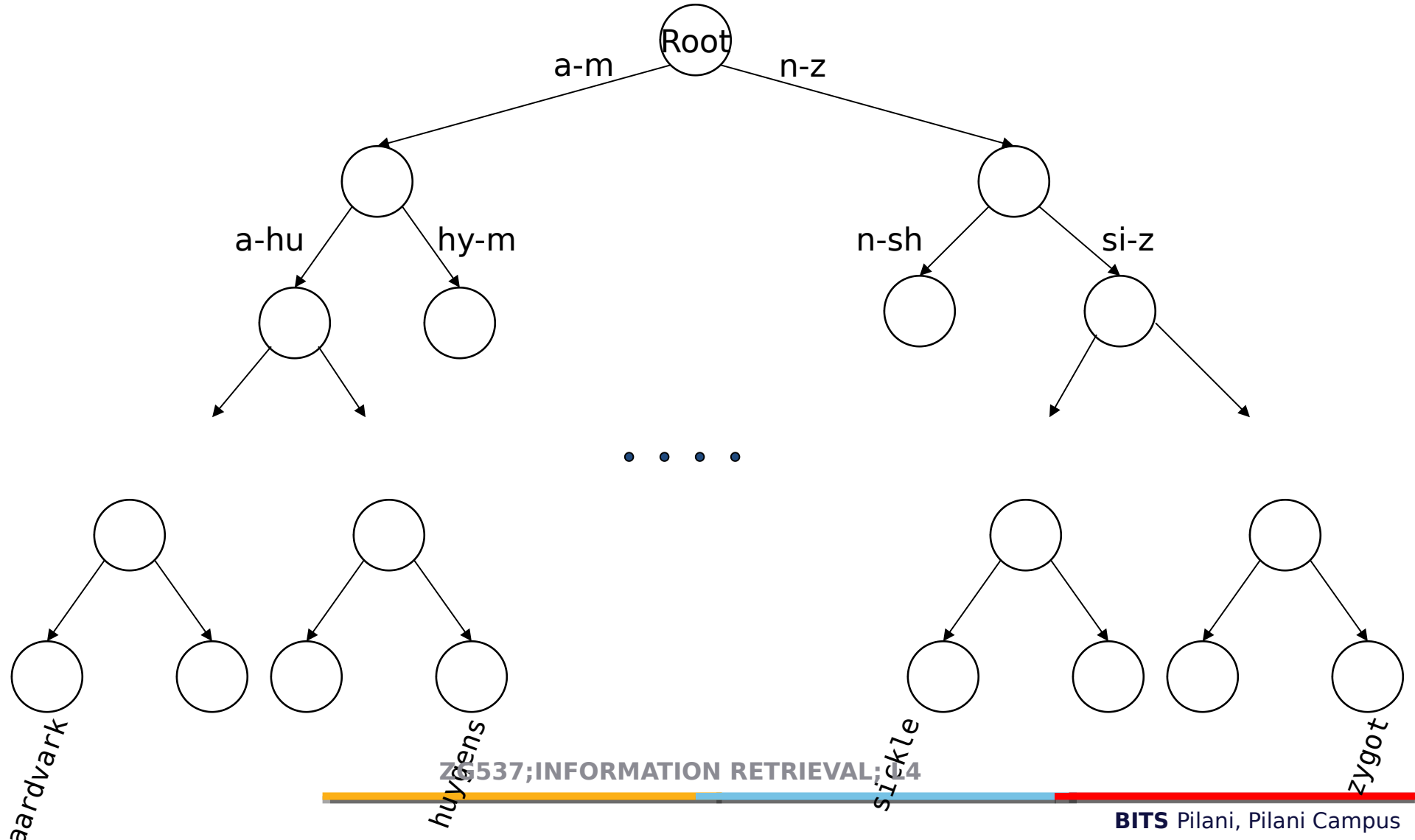
Hashtables

- Each vocabulary term is hashed to an integer
- **Pros:**
  - Lookup is faster than for a tree: O(1)
- **Cons:**
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Trees

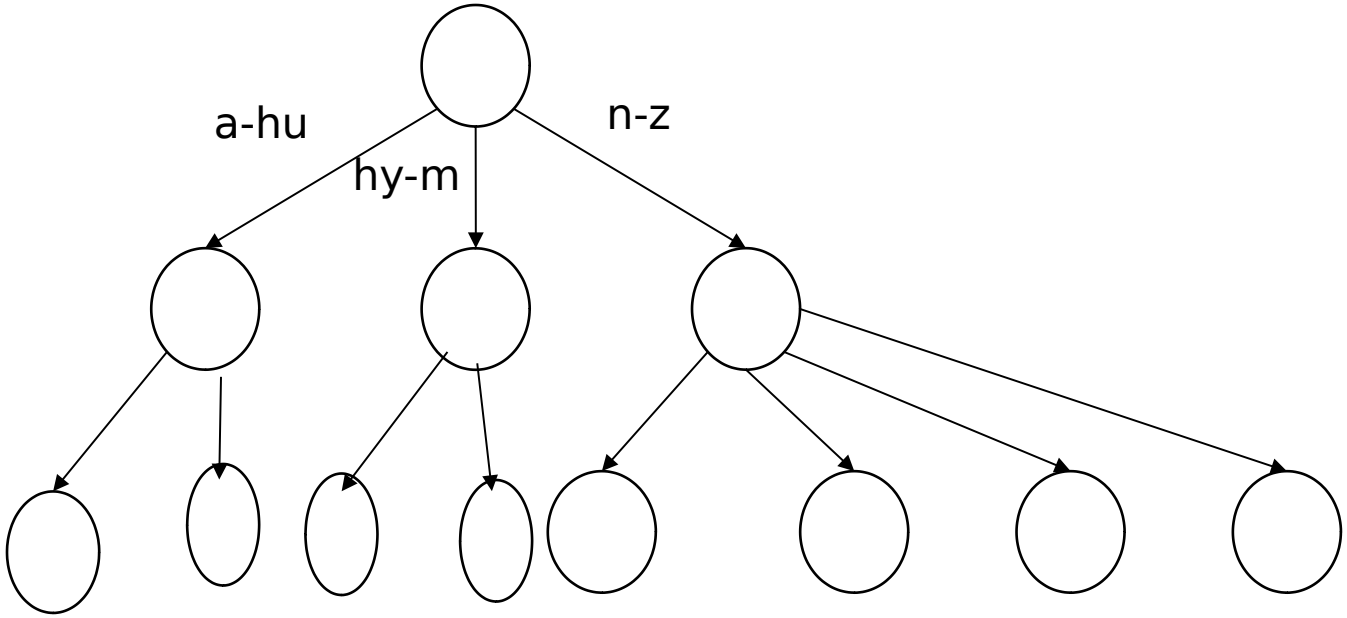
- Trees solve the prefix problem (find all terms starting with automat).
- Simplest tree: binary search tree
- Search is slightly slower than in hashes: O(log M), where M is the size of the vocabulary.
- O(log M) only holds for balanced trees

Tree: Binary Search Tree



Tree: B-tree

- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a, b* are appropriate natural numbers, e.g., [2,4].



Trees

Pros:

- Solves the prefix problem (terms starting with *hyp*)

Cons:

- Slower: O(log *M*) [and this requires *balanced* tree]
- Rebalancing binary trees is expensive
  - But B-trees mitigate the rebalancing problem

Tries and FSTs

innovate achieve lead

Tries (or Prefix Trees) and advanced variants like Finite State Transducers **(FSTs**).

- Terms are stored based on their prefixes and shared characters.
- FSTs are highly compressed versions of Tries.

Pros:

- Efficient Prefix Search: Excellent for supporting prefix and auto-complete queries.
- Space Saving: Can save significant space by sharing common prefixes.
- Time complexity in Tries is proportional to length of terms :O(L)

Tolerant Retrieval

innovate achieve lead

- **Wild-card queries**
- Spelling correction
- Phonetic correction

Wild-card queries

- Uncertain of the spelling of a query term (e.g., Sydney vs. Sidney, which leads to the wildcard query S*dney)
- Seeking documents containing the variants (e.g., color vs. colour)
- Seeking documents containing variants of a term that would be caught by stemming, but is unsure whether the search engine performs stemming (e.g., judicial vs. judiciary, leading to the wildcard query judicia*)
- Uncertain of the correct rendition of a foreign word or phrase (e.g., the query Universit* Stuttgart).

Wild-card queries: *

- ***mon*:*** (trailing wildcard query) find all docs containing any word beginning with "mon".
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo (w is the set of terms which start with mon)***

Wild-card queries: *

- ***mon:*** (leading wildcard query) find words ending in "mon":
  - Maintain an additional B-tree (reverse B-tree)
  - Reverse B-tree is one in which each root-to-leaf path of the B-tree corresponds to a term in the dictionary written backwards:
  - Can retrieve all words in range: ***nom ≤ w < non***.

Wild-card queries: *

- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ****tion*** in a B-tree and the reverse B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s occur at the end
- This gives rise to the **Permuterm** Index.

Wild-card queries: *

How to process the query ***se*ate** *AND* ***fil*er ?***

Wildcard Query Processing

- **Permuterm indexes**
- **K-gram indexes**

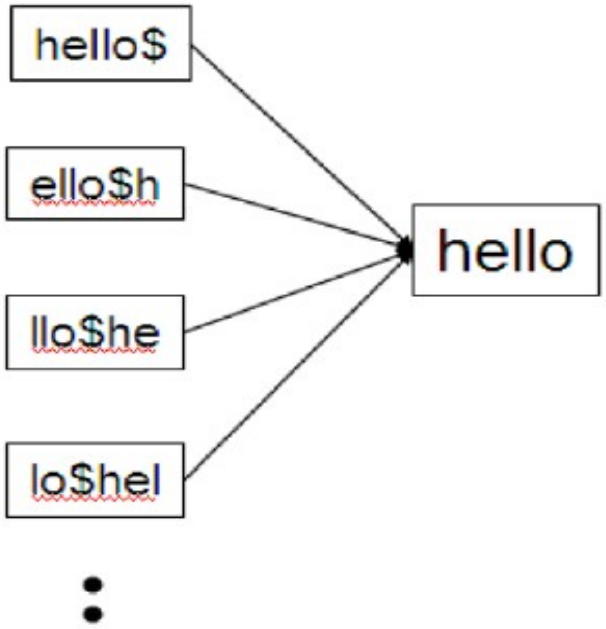
Both techniques share a common strategy:

- Express the given wildcard query q_w as a Boolean query Q on a specially constructed index, such that the answer to Q is a superset of the set of vocabulary terms matching q_w.
- Then, we check each term in the answer to Q against q_w, discarding those vocabulary terms that do not match q_w

Permuterm indexes

- Introduce a special symbol $ into the character set, to mark the end of a term
- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell, $hello***
  - where $ is a special symbol.

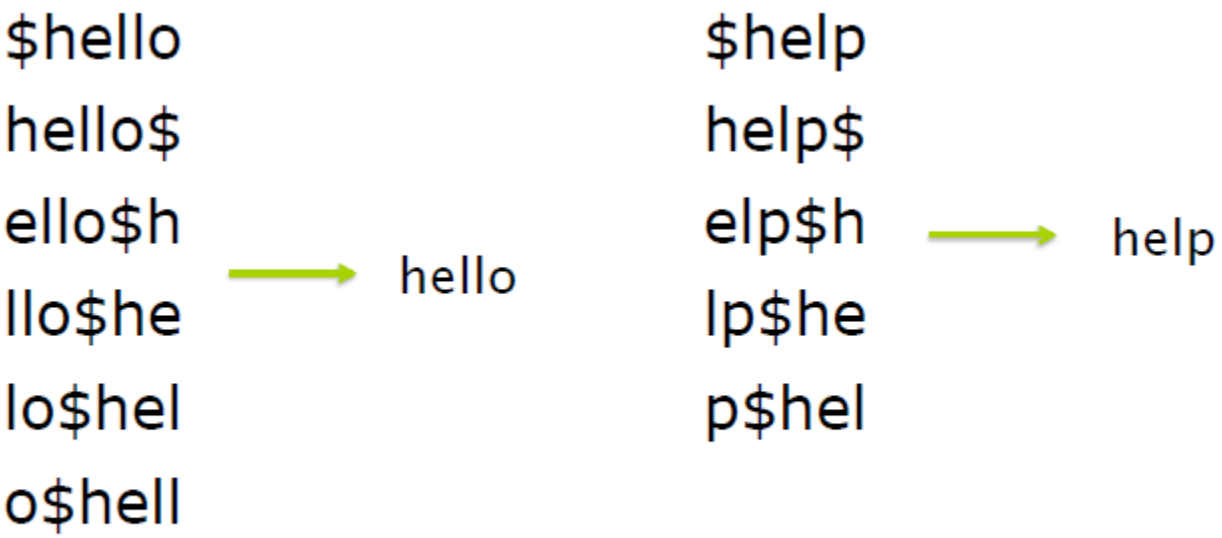
Permuterm indexes



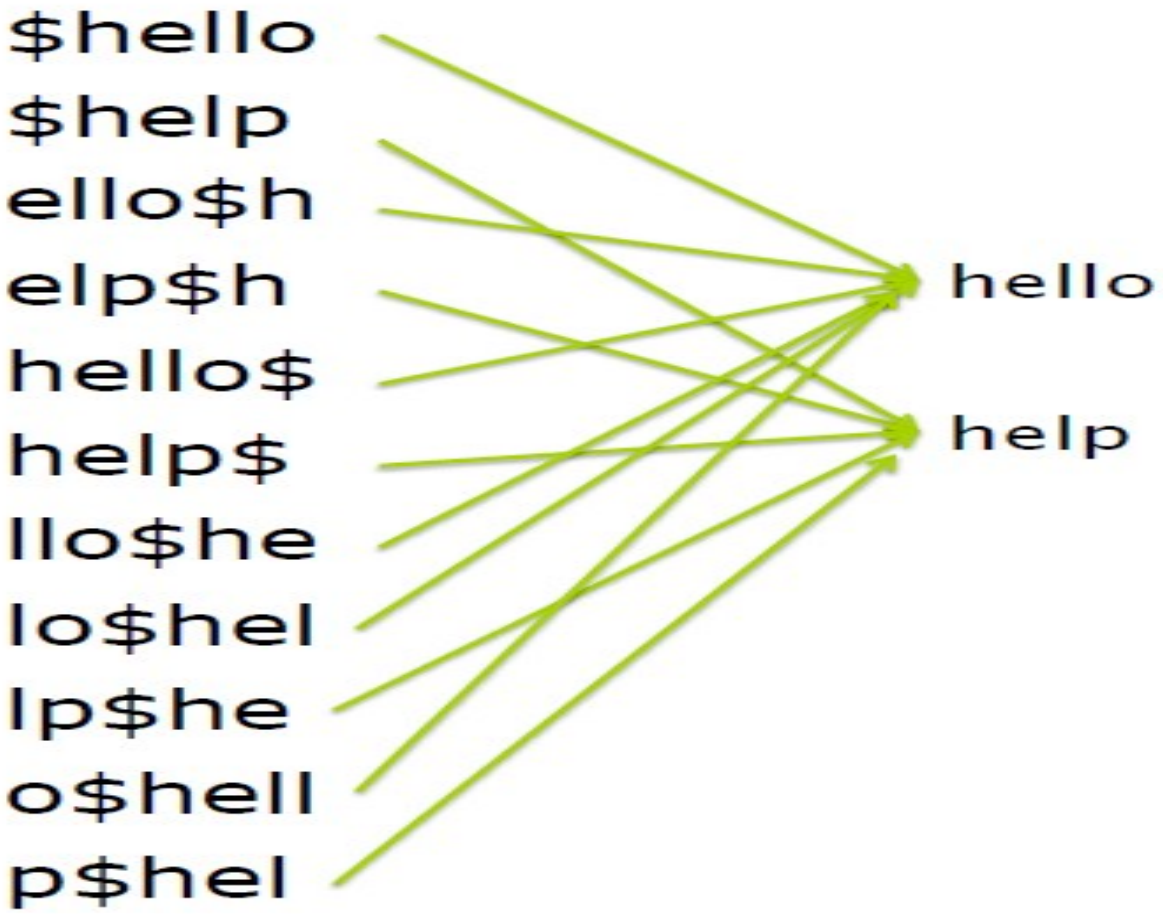
► Figure 3.3 A portion of a permuterm index.

- From the permuterm we can get the term and then from the standard index we can get the documents containing the term.

Permuterm indexes



Permuterm indexes



Permuterm indexes

- Consider the wildcard query m*n.
- Rotate this wildcard query so that the * symbol appears at the end of the string after adding the $ symbol
- Rotated wildcard query becomes n$m*
- Look up this string in the permuterm index and get the terms
- Search in the inverted index all the documents indexed by these terms
- M*n$, *n$m, n$m*
- H*o, o$h*

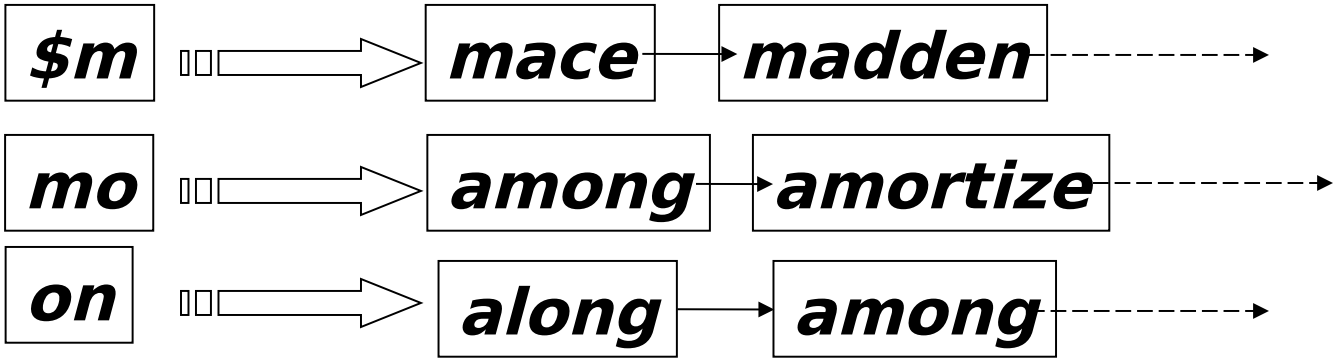
k-gram indexes

innovate achieve lead

- A *k-gram* is a sequence of *k* characters.
- cas, ast and stl are all 3-grams occurring in the term castle.
- The full set of 3-grams generated for castle is: $ca, cas, ast, stl, tle, le$
- In a *k-gram index*, the dictionary contains all *k*-grams that occur in any term in the vocabulary
- Each postings list points from a *k*-gram to all vocabulary terms containing that *k*-gram

k-gram indexes

- The *k*-gram index finds *terms* based on a query consisting of *k*-grams (here *k*=2).



- Maintain a *second* inverted index *from bigrams to dictionary terms* that match each bigram.

k-gram indexes

- Query ***mon**** can now be run as
  - ***$m** AND **mo** AND **on***
- Gets terms that match AND version of the wildcard query
- But the enumerated term ***moon*** will also be retrieved (False Positive)
- Must post-filter these terms against query
- Surviving enumerated terms are then looked up in the term-document inverted index
- Fast, space efficient.

- mo*n

Exercise - Permuterm Index

innovate achieve lead

Exercise 3.2

Write down the entries in the permuterm index dictionary that are generated by the term mama.

Exercise 3.3

If you wanted to search for s*ng in a permuterm wildcard index, what key(s) would one do the lookup on?

Tolerant Retrieval

innovate achieve lead

- Wild-card queries
- **Spelling correction**
- Phonetic correction

Spelling correction

- Among the various alternative correct spellings for a mis-spelled query, choose the “nearest” one.
- When two correctly spelled queries are tied (or nearly tied), select the one that is more common.

Two forms of Correction:

- Isolated word
  - Check each word on its own for misspelling
  - Will not catch typos resulting in correctly spelled words
  - e.g., ***from → form***
- Context-sensitive
  - Look at surrounding words,
  - e.g., ***I flew form Heathrow to Narita***

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this:
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.

      (Including the mis-spellings)

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *k*-gram index

Edit distance

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
  - from ***cat*** to ***dog*** is 3.
- Dynamic programming technique is used for computing the edit distance

Levenshtein Distance: Algorithm

Note: To understand this algorithm, Dynamic programming design technique understanding is required.

- The algorithm fills the entries in a matrix *m* whose two dimensions equals the length of the two strings, whose edit distances are being computed
- The (*i*, *j*) entry of the matrix will hold the edit distance between the strings consisting of the first *i* characters of *s*1 and the first *j* characters of *s*2.

Levenshtein Distance: Algorithm

LEVENSHTEINDISTANCE(s_1, s_2)

```
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|s1|, |s2|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein Distance: Algorithm

innovate achieve lead

LEVENSHTEINDISTANCE(s_1, s_2)

```
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|s1|, |s2|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein Distance: Algorithm

innovate achieve lead

LEVENSHTEINDISTANCE(s_1, s_2)

```
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|s1|, |s2|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein Distance: Algorithm

innovate achieve lead

```
LevenshteinDistance(s1, s2)
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|s1|, |s2|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Levenshtein Distance: Algorithm

LEVENSHTEINDISTANCE(s_1, s_2)

```
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]}
 9            else  m[i, j] = min{m[i-1, j]+1, m[i, j-1]+1, m[i-1, j-1]+1}
10  return m[|s1|, |s2|]
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

Each cell of Levenshtein matrix

Cost of getting here from my upper left neighbour (by copy or replace)	Cost of getting here from my upper neighbour (by delete)
Cost of getting here from my left neighbour (by insert)	Minimum cost out of these

Levenshtein Distance: Example

		f		a		s		t	
	0	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**4**
c	**1** **1**	*1* *2*	*2* *1*	**2** 3 **2** **2**		**3** 4 **3** **3**		**4** 5 **4** **4**	
a	**2** **2**	**2** **2** 3 **2**		*1* *3* *3* *1*		*3* *4* *2* *2*		4 5 **3** **3**	
t	**3** **3**	**3** **3** 4 **3**		3 **2** 4 **2**		*2* *3* *3* *2*		*2* *4* *3* *2*	
s	**4** **4**	**4** **4** 5 **4**		4 **3** 5 **3**		**2** 3 4 **2**		*3* *3* *3* *3*	

Levenshtein Distance: Example

		f	a	s	t
	0	1	2	3	4
c	1	1	2	3	4
a	2	2	1	2	3
t	3	3	2	2	2
s	4	4	3	2	3

Levenshtein Distance: Example

		s	n	o	w
	0	1 1	2 2	3 3	4 4
o	1 / 1				
s	2 / 2				
l	3 / 3				
o	4 / 4				

Levenshtein Distance: Example

		s		n		o		w	
	0	1	1	2	2	3	3	4	4
o	1 / 1	1 / 2	2 / 1	2 / 2	3 / 2	2 / 3	4 / 2	4 / 3	5 / 3
s	2 / 2	1 / 3	2 / 1	2 / 2	3 / 3	3 / 3	3 / 4	3 / 4	4 / 3
l	3 / 3	3 / 4	2 / 2	2 / 3	3 / 2	3 / 3	4 / 3	4 / 4	4 / 4
o	4 / 4	4 / 5	3 / 3	3 / 4	3 / 3	2 / 4	4 / 2	4 / 3	5 / 3

Levenshtein Distance: Example

		s	n	o	w
	0	1	2	3	4
o	1	1	2	3	4
s	2	1	3	3	3
l	3	3	2	3	4
o	4	3	3	2	3

Weighted edit distance

- Same as edit distance, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors Example: ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Using edit distances

- Given query, first enumerate all character sequences within a preset edit distance (e.g., 2)
- Intersect this set with list of “correct” words
- Show terms you found to user as suggestions
- Alternatively,
  - look up all possible corrections in the inverted index and return all docs -- slow
  - run with a single most likely correction

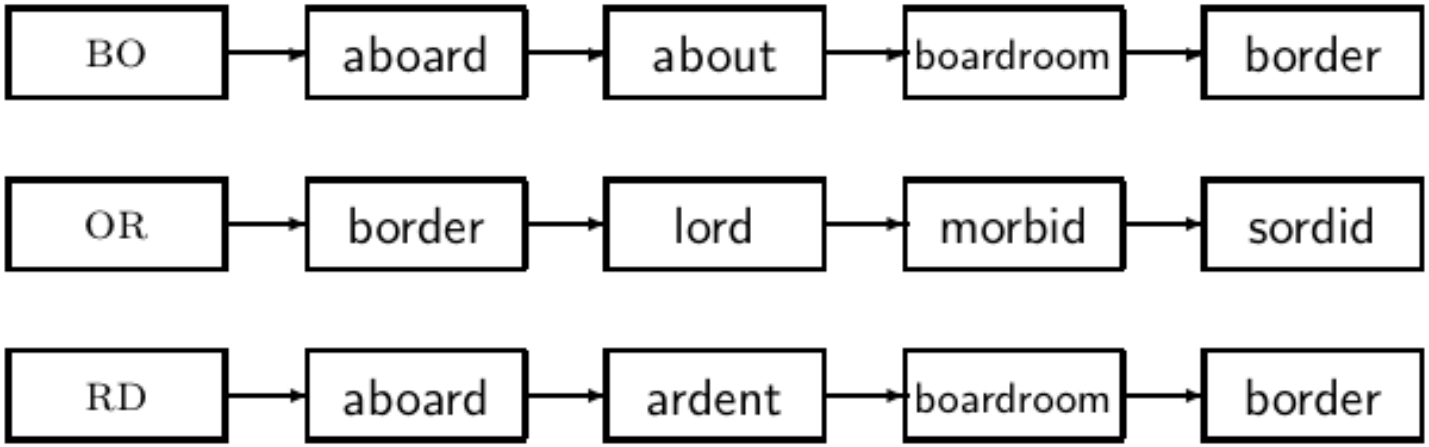
Using edit distances

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative?
- One possibility is to use ***k*-gram overlap**.
  - to cut the set of candidate dictionary terms

k-gram index

- Enumerate all the k-grams in the query string as well as in the document
- Use the k-gram index to retrieve all terms matching any of the query k-grams
- Threshold by number of matching k-grams
- Example: bigram index, misspelled word **boardroom** **as** **bordroom**
- Bigrams: ***bo, or, rd, dr, ro, oo, om***

k-gram index


BO
aboard
about
boardroom
border
OR
border
lord
morbid
sordid
RD
aboard
ardent
boardroom
border

k-gram index

Example with trigrams

- Suppose the text is **November**
  - Trigrams are nov, ove, vem, **emb, mbe, ber.**
- The query is **december**
  - Trigrams are dec, ece, cem, **emb, mbe, ber.**
- So 3 trigrams overlap (of 6 in each term)
  - **normalized measure of overlap?**

k-gram index

- **Jaccard coefficient**
- A commonly-used measure of overlap
- Let X and Y be two sets; then the J.C. is **X ∩Y / X ∪Y**
- Equals 1 when X and Y have the same elements and zero when they are disjoint
- X and Y don t' have to be of the same size
- Always assigns a number between 0 and 1
  - Now threshold to decide if you have a match
  - E.g., if J.C. > 0.8, declare a match

Context-sensitive spell correction

- Take the Example : *flew **form** munich*
  - How can we correct ***form*** here?
- One idea: hit-based spelling correction
  - Retrieve “correct” terms close to each query term *for "flew form munich":*
  - *flea for flew, from for form, munch for munich*
  - Now try all possible resulting phrases as queries with one word “fixed” at a time
  - Try query *“flea form munich”*
  - Try query *“flew from munich”*
  - Try query *“flew form munch”*
- The correct query *“flew from munich”* has the most hits.

Context-sensitive spell correction

Another approach

- Break phrase query into a conjunction of biwords.
- Look for biwords that need only one term corrected.
- Enumerate phrase matches and rank them.

Tolerant Retrieval

innovate achieve lead

- Wild-card queries
- Spelling correction
- **Phonetic correction**

Phonetic correction

- Misspellings that arise because the user types a query that sounds like the target term
- The main idea here is to generate, for each term, a “phonetic hash” so that similar-sounding terms hash to the same value
- Algorithms for such phonetic hashing are collectively known as ***soundex*** algorithms.

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** → ***tchebycheff***
- Invented for the U.S. census … in 1918

Soundex – algorithm

- Turn every token to be indexed into a 4-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index
- Do the same with query terms
- When the query calls for a soundex match, search this soundex index.

Soundex – algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

B, F, P, V → 1

C, G, J, K, Q, S, X, Z → 2

D,T → 3

L → 4

M, N → 5

R → 6

Soundex – algorithm

4. Remove one out of each pair of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

What will be the soundex code for Hermann? H065055

H06505—H655

Skill

Square

Preparation

Accommodation

~~Skill – S2044- S204- S24- S240~~

Skill – S2044- S04- S4-S400

Preparation – P6010603005- P61635-P616

Accommodation---A220550303005---A2050303005-A25335-A253

Soundex

- Soundex is the classic algorithm, provided by most databases (Oracle, Microsoft, …)
- Not much useful for information retrieval
- Okay for “high recall” tasks (e.g., Interpol), though biased to names of certain nationalities

Exercise

- Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about
- Identify some of the key design choices in the index pipeline:
  - Does stemming happen before the Soundex index?
  - What about *n*-grams?
- Given a query, how would you parse and dispatch sub- queries to the various indexes?

Exercise

- Draw yourself a diagram showing the various indexes in a search engine incorporating all the functionality we have talked about
- Identify some of the key design choices in the index pipeline:
  - Does stemming happen before the Soundex index?
  - What about *n*-grams?
- Given a query, how would you parse and dispatch sub- queries to the various indexes?

Resources

innovate achieve lead

Wild card search in Google

Edit distance demo

Wild card search in Google –Article

Soundex Demo1

Soundex Demo2

Resources

Efficient spell retrieval:

K. Kukich. Techniques for automatically correcting words in text. ACM Computing Surveys 24(4), Dec 1992.

J. Zobel and P. Dart.Finding approximate matches in large lexicons. Software - practice and experience 25(3), March 1995. http://citeseer.ist.psu.edu/zobel95finding.html

Mikael Tillenius: Efficient Generation and Ranking of Spelling Error Corrections. Master's thesis at Sweden's Royal Institute of Technology. http://citeseer.ist.psu.edu/179155.html

Nice, easy reading on spell correction:

Peter Norvig: How to write a spelling corrector

http://norvig.com/spell-correct.html

THANK YOU